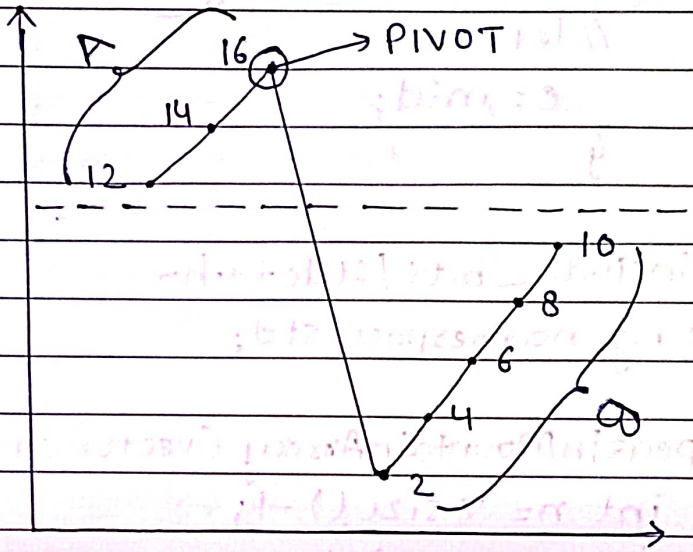
18/09/2023
Monday

* Pivot Index in rotated and sorted array →

I/P = 12 14 16 2 4 6 8 10

O/P = Pivot Index is: 2



Logic→ // corner case → when array contains single element

```
if (s==e) {
    return s;
}
```

[We can return either e or mid too.]

// The two cases is for → where sequence breaks i.e. | 16 | 2 |

```
if (mid+1<n && v[mid]>v[mid+1]){
    return mid;
}
```

mid+1 <n because mid+1 can be gone out of range of array if doesn't mentioned.

```
if (mid-1>=0 && v[mid]<v[mid-1]){
    return mid-1;
}
```

mid-1 >=0, so that it does not go out of the range of array.

// These two cases when mid is in either part A or part B.

```
if (v[s] > v[mid]){
    e = mid-1;
}
```

In part B, every element must be smaller than first element in part A i.e. start index. That's why the condition v[s] > v[mid] came.

So, to approach pivot element, move to the left.

```
if (v[s] < v[mid]){
    s = mid+1;
}
```

In part A, every element must be greater than first element i.e. start index.

So, to approach pivot element, move to the right.

- Update mid.
- Else if → nothing in the above conditions is true, return -1.

__/__/____

* Program → Pivot Index

```
#include<bits/stdc++.h>
using namespace std;

int pivotIndex (vector<int>&v){
    int n=v.size();
    int s=0; int e=n-1;
    int mid= s+(e-s)/2;
  while (s<=e){
     if (s==e){
        return s;
      }
    else if (mid+1<n && v[mid]>v[mid+1]){
        return mid;
      }
    else if (mid-1>=0 && v[mid]<v[mid-1]){
        return mid-1;
      }
    else if (v[s]>v[mid]){
        e=mid-1;
      }
    else
        s=mid+1;
    }
mid= s+(e-s)/2; }
return -1;
}
int main(){
 vector<int>v{12,14,16,2,4,6,8,10};
   int pivot = pivotIndex(v);
   cout<<"pivot Index is: "<<pivot<<endl;
}
```

__/__/__

* **Search in Rotated and Sorted Array →**

I/P = 12 14 16 2 4 6 8 10
target = 6

O/P = target found at index : 5

logic →

- Make a function to find the pivot index in rotated and sorted array.
- Now, the array is divided into part A and part B.
- Make a function for binary search. It depends on target whether to find in part A or part B.
- Make one more function. This function will tell whether to apply binary search in part A or part B. In this function →
  - → call pivot index function.
  - → Initialise a variable ans with -1. This ans variable will store the target if present otherwise it will return -1.
  - → // search in A →

```
if (target >= v[0] && target <= v[pivot])
    ans = binarySearch(v, 0, pivot, target)
```

  - → // search in B →

```
else
    ans = binarySearch(v, pivot+1, n-1, target)
```

  - → return ans.

Program→

__/__/____

```
#include<bits/stdc++.h>
using namespace std;

// this function is for finding the pivot index
// during search in rotated & sorted array.
int pivotIndex(vector<int>&v){
    int n=v.size();
    int s=0; int e=n-1;
    int mid=s+(e-s)/2;
    while(s<=e){
        if(s==e){
            return s;
        }
    else if(mid+1<n && v[mid]>v[mid+1]){
        return mid;
        }
    else if(mid-1>=0 && v[mid]<v[mid-1]){
        return mid-1;
        }
    else if(v[s]>v[mid]){
        e=mid-1;
        }
    else{
        s=mid+1;
        }
    mid=s+(e-s)/2;
    }
return -1;
}
```

__/__/____

```
// this function is for binary search in array
// it depends on target whether to apply
// in part A or part B.
int BinarySearch(vector<int>&v, int s, int e, int target)
{
    int mid = s + (e-s)/2;
    while (s<=e){
      if (v[mid] == target){
          return mid;
        }
      else if (target > v[mid]){
            s = mid+1;
          }
      else{
            e = mid-1;
        }
      mid = s + (e-s)/2;
    }
return -1;
}
// this function is for finding whether binary search
// to be apply on part A or part B.
int Search(vector<int>&v, int target){
    int pivot = pivotIndex (v);
    int n = v.size();
    int ans = -1;
  // search in A
  if (target >= v[0] && target <= v[pivot]){
      ans = BinarySearch(v, 0, pivot, target);
    }
```

__/__/____

// search in B

```
    else {
        ans = BinarySearch(v, pivot + 1, n - 1, target);
    }
return ans;
}

int main(){
    vector<int>v{12,14,16,2,4,6,8,10};
    int target = 6;
    int targetIndex = Search(v, target);
    if (targetIndex == -1){
        cout<< "target not found" <<endl;
    }
    else
        cout<< "target found at index:" <<targetIndex
                                        <<endl;
```

* **Square Root of a number (Return only integer)**

I/P = 129

O/P = 11

Terms →

= **Search space** → The given range where the answer must lies.

= **Predicate function** → It returns true or false accordingly the given xyz condition.

Logic → • if (mid * mid == x){
return mid;
}

This is the case when mid is the square root of given input x.

• else if (mid * mid < x){
ans = mid;
s = mid+1;
}

In this case, store the mid in ans variable and move to the right.

1 2 3 4 5 6 7 8 9 10 11 12 13 ---- 129

True false

The range for which predicate function is returning true, that range of elements should be stored.

__/__/____

- else {
    e = mid - 1;
  }

This is the case when mid * mid > x. Here, predicate function returns false. So, nothing to be stored, just move to the left.

- Update mid and return ans.

Program→

```
#include<bits/stdc++.h>
using namespace std;
int sqrt(int &x){
    int s=0; int e=x; (// search space = 0->x)
long long int mid= s+(e-s)/2; (// long long int because mid*mid will give large value which may cross range of int)

while(s<=e){
    if(mid*mid==x){
        return mid; }
    else if(mid*mid<x){
        ans=mid;
        s=mid+1; }
    else{
        e=mid-1; }
    mid= s+(e-s)/2;
}
return ans;
}
int main(){
    int x=129;
    int sq_root = sqrt(x);
    cout<<"square root is:"<<sq_root<<endl;
}
```

__/__/____

* 2-D to 1-D conversion → c * i + J

* 1-D to 2-D conversion →

rowNo = index / col

columnNo = index % col

* Binary Search in 2-D matrix →

I/P = 1 2 3 Target = 8
4 5 6
7 8 9

O/P = target found or not : 1 (1 represents true)

logic → As 2-D matrices are also stored as 1-D i.e. linearly in the memory.

So, we have to find the conversion of 1-D into 2-D as per formula →

row_Index = mid / col

col_Index = mid % col

Initialise a variable currNumber = v[row_Index][col_Index].

This currNumber will store the current position

```
if (currNumber == target) {
        return true;
    }
else if (target > currNumber){
        s = mid + 1;
    }
else    e = mid - 1;
}
```

* Program →

```
#include <bits/stdc++.h>
using namespace std;

bool binarySearch(vector<vector<int>>&v, int target){
        int row= v.size();
        int col = v[0].size();
        int n= row*col;
        int s =0;
        int e =n-1;
        int mid= s+(e-s)/2;
       while (s<=e) {
       int row_Index = mid /col;
       int col_Index = mid %col;
       int currNumber = v[row_Index][col_Index];
       if (currNumber ==target){
                return true;
              }
       else if ( target > currNumber){
                s= mid+1;
              }
       else {
            e=mid-1;
          }
     mid= s+(e-s)/2;
    }
return false;
}
int main(){
 vector<vector<int>>v{{1,2,3},{4,5,6},{7,8,9}};
    int target = 8;
    cout << "target found or not: "<< binarySearch
                                    (v,target)<<endl;
```